Modeling Computer Programs: A Case Study in Predicting Variable Types via Graph Neural Nets

Project URL: `https://github.com/shashank-srikant/6.867_term_project/`

Alex Renda, Katie Lewis, Shashank Srikant
CSAIL, MIT
{renda, kmlewis, shash}@mit.edu

Abstract

We explore the problem of modeling computer programs to infer their properties. We argue that programs can be modeled as Markov Random Fields, where the latent random variables in the graph represent the properties of interest. We approximate inference over such a graphical model via graph neural networks (GNN), a neural network architecture which captures graph-like structural information. As a case study, we work on the problem of inferring variable types in JavaScript programs. We show that modeling this task using GNNs provides a 10-15% improvement in accuracy of type prediction when compared to a bi-directional RNN baseline. Via ablation studies, we also investigate the benefits of various modeling assumptions we make in our GNN construction.

1 Introduction

Inferring properties of computer programs is a task central to the programming languages (PL) community. Although many challenges in the field are undecidable, like figuring out whether a program terminates or finding a test case which crashes a program, the program analysis community has designed algorithms and techniques that approximate solutions to such undecidable problems. These techniques have been explored to an extent that working solutions have been shipped in real-time, commercial products.

One such property of interest is ascertaining the types of variables used in programs. Static, strongly-typed languages provide considerable guarantees at compile time, enabling programmers to write more correct and interpretable code. Although the advantages of this paradigm are well understood by the programming community, many popular languages such as Python, Ruby, and Javascript still do not enforce static typing. These languages prevent typing errors at run time, and do minimal (if any) checking at compile time. This approach of dynamic typing is referred to as *duck typing* ("If it walks like a duck and it quacks like a duck, then it must be a duck"). Despite the lack of static assurances given by duck typing, it is widely adopted and is used extensively in production systems. This calls for some ability to infer, at compile time, the type of variables and expressions in the program, to statically rule out bugs while still enjoying the usability benefits of duck typing.

One recent approach to this inference problem is *gradual typing* [1], which allows for partial specification of type annotations in an otherwise untyped language. This enables established type-inference algorithms to determine types of other variables used in a program, even ones that do not have explicit annotations. Another approach, which is the subject of this work, is to infer types statistically. The problem of inferring types can be cast as a supervised learning problem, where a model can be trained on a corpus of programs with identified types in order to be able to infer types of unseen variables in unseen programs.

This statistical approach has been validated with a modest degree of success in previous work [2]. Allamanis *et al.* modeled the problem as a sequence translation task, and trained bi-directional RNNs to predict types (see Section 3). In this work, we ask whether there are inductive biases which are unique to this input space, computer programs, and how they can be exploited in such a prediction task. We motivate this problem as an inference task over a probabilistic graphical model, and approximate it using a graph-based neural network. We show that this modeling assumption outperforms sequence-based approaches for type inference.

2 Motivation & Background

Modeling programs. With the advent of large public software repositories like GitHub, and better performing machine learning tools like deep neural networks, the problem of determining good statistical models for programs has recently become an active research area. This problem primarily consists of determining some latent representation of a program, and then using that representation to perform some sort of analysis of the program. Several inference tasks in this domain have seen moderate success. These include predicting meaningful variable names [3], detecting bugs and vulnerabilities [4], grading programs [5], and others. Each of these approaches models the problem as a supervised learning task, where features are extracted from programs and are used to predict the property of interest. This is often a lossy step, where the rich structural and semantic information present in programs is only roughly approximated via these features. The community therefore focuses much energy on choosing the right models to capture the rich structural information of programs. Very recent work has explored using probabilistic graphical models, and graph neural networks to model problems inherently containing graph-like structures (e.g. synthesis of chemical compounds [6]).

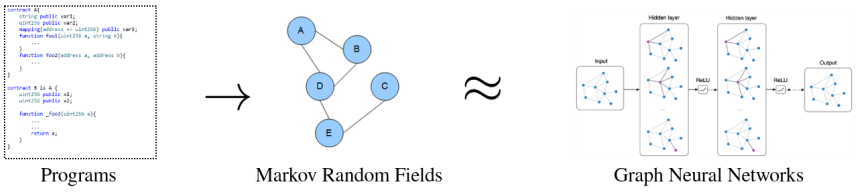

Figure 1: **Summary of our work.** We motivate the problem of modeling programs as Markov Random Fields. We argue that a graph neural network approximates inference over a MRF. Specifically, we investigate the problem of inferring types in JavaScript programs. This problem formulation was first put forward by Allamanis *et al.* [2].

The work in this paper is inspired by such an approach: we lay out an argument for why probabilistic graphical models are an appropriate tool for modeling this problem, exploring the performance of statistical type inference under this modeling assumption.

Abstract Syntax Trees (AST) & other program representations. A program can be represented in many ways: as the stream of tokens in the source code file, as intermediate graphs generated by a compiler, or as an embedding informed by any of the previous representations. Our representation is based on the Abstract Syntax Tree (AST) of a program, which is the tree that shows how that program could be generated by the formal grammar defining the programming language. For example, the AST of an expression `1 + 2` might have the binary operator + at the root, with `1` and `2` as its left and right children respectively. The AST of a program concisely represents many of the locality properties we would desire in a program representation: for instance, in the sequence of statements `x := 1 + 2; y := 3`, the `2` would be closer to the `x` in the AST than it would be to the `y` (despite the layout of the source code).

Type inference. The statistical type inference problem we solve in this paper is not the only type inference algorithm. There exist deterministic algorithms like Hindley-Milner's W-algorithm (HM) to infer and check the types of variables in programs. HM assumes some known structure between how variables interact, and inductively applies type-propagation rules. This works well in practice for languages which have been designed keeping such analyses in mind, like OCaml. With languages like JavaScript and Ruby though, implementing HM will result in poor specificity, with most types simply being inferred as *any* (i.e. undeterminable). Such inference is not helpful to an end-user, and hence warrants other techniques which may prove effective in discovering a more specific type.

We therefore explore machine learning as a tool to help learn these types. The key insight is that the problem of inferring types involves two sub-tasks: (i) finding a signal which can inform the type of a variable, and (ii) modeling a structured network which can help propagate this signal to other occurrences of the same variable, or variables that are similar to it.

Graph neural networks. Graph-structured input modalities to machine learning problems have been explored over the last decade in light of deep neural network architectures. LeCun *et al.* published a position paper [7] which describes learning on graph-structured input modalities as a convolutional neural network learning non-Euclidean filters. DeepMind, with its recent position paper on inductive biases captured by graph neural networks [8], has initiated a resurgence of interest in Graph Neural Networks (GNNs).

Broadly, a GNN is set up as follows: each node and edge in the graph has an associated learned embedding as its initial state. Hidden layers connect nodes to edges, and vice versa. A GNN is run for a number of iterations, similar to a recurrent neural network. Each iteration of the GNN follows a message passing algorithm: an edge from node A to

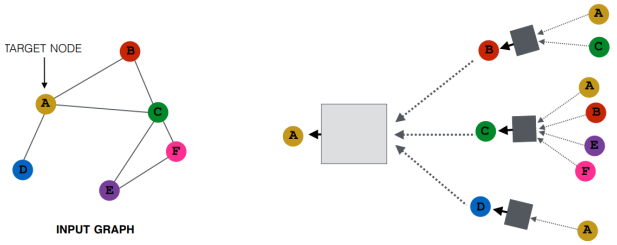

Figure 2: A typical graph neural network architecture, from [9]. The key idea is to generate node embeddings based on neighborhood information.

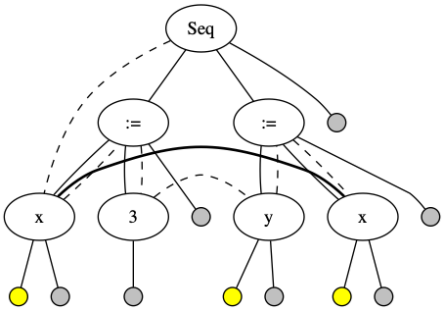

Figure 3: Constructed Markov Random Field for an example program: `x := 3; y := x;` Shaded nodes are observed, highlighted are desired posteriors

B computes its new hidden state as a function of the edge's old state, and the state at node A. Next, each node computes its new embedding as a function of its old embedding and the aggregated embedding of each incoming edge. Figure 2 illustrates this architecture.

This general idea is cast in multiple ways: LeCun portrays GNNs as CNNs with non-Euclidean filters; DeepMind's portrays them as auto-encoder model over graphs; Microsoft portrays them as gated recurrent networks. The generality of the framework leaves it open to many valid interpretations.

Probabilistic interpretation. Having introduced the problem of type inference, and graph neural networks, we lay out a justification in this section for why such a network is the right form of inductive bias to model programs.

A Markov Random Field (MRF) is an undirected graphical model with the Markov assumption that a node is conditionally independent from all other nodes given its neighbors [10]. If we were to construct a Markov Random Field with an unobserved latent variable for each of our AST nodes, add the various (undirected) edges described in Section 3.2, tag each node in the graph with an additional random variable (the observed value representing the node's AST type), and tag each program variable with an additional random variable representing its type, we would get a graph that looks roughly like the one in Figure 3. We could then perform posterior inference on the type of the program variable nodes of this graph using belief propagation [11], which gives an approximate solution to the true posterior [12], giving our desired type predictions.

We specifically argue the following two points: (i) the induced MRF is the correct model for this problem, and (ii) the graph neural network performs posterior inference on this network.

For point (i), we believe that the classes of edges we use roughly capture the extent of conditional dependence in the model, in that the latent state of a variable (that predicts its type) is roughly conditionally independent from all others given its neighbors along those edges. This full justification is left to Section 3.2, but it is summarized by saying that we add edges between nodes if those nodes are local in some sense (whether that means they are colocated in the code, or deal with similar variables). Some edges we added may be spurious, and while this may make it harder to perform inference, the MRF with too many edges can just be viewed as a refinement of the MRF with too few edges. This is also the case with the undirectedness assumption, where it's likely that some of the edges would ideally be represented as directed edges. However, these violated assumptions are also present when using MRFs for images or other applications, and while the assumptions can be shown to be incorrect, MRFs in those domains still prove to be useful models [13].

Point (ii) requires arguing that our graph neural network architecture approximates posterior inference on the network. This argument is much easier: belief propagation, a message passing algorithm, approximates the correct posterior on graphs with loops [12]. GNNs are essentially a message passing algorithm, where the messages are the result of some function learned by the GNN on the current latent state of a given node. By merit of being neural networks, GNNs are capable of approximating any function [14] (and storing latent state), meaning that GNNs are capable of learning belief propagation (assuming we run the message passing for enough iterations). Therefore, assuming that training our GNN finds its global optimum, we are guaranteed a solution at least as good as the approximate solution of belief propagation on the MRF, meaning that a GNN should be capable of solving the type inference problem.

3 Approach

Our approach to the type prediction problem is to apply a GNN to a graph constructed from each TypeScript program, where the graph encodes much of the information that may be relevant in trying to predict a given variable's type.

3.1 Dataset

We train and evaluate on the dataset from the DeepTyper project [2], which contains 1000 popular open-source TypeSript projects from GitHub. Each project was parsed with the TypeScript compiler, which infers type information (including the all-encompassing `any` type as a fallback) for all occurrences of each identifier. To make learning tractable, we randomly sampled a total of 90 projects from the dataset. Dataset statistics can be seen in Table 1. For our prediction task, to avoid the sparsity problems that come with such a

	Total	Train	Test
# TS projects	90	72	18
# Source files	3,990	3,025	965
# Unique types	20	20	20
# Unique AST nodes	60	60	60
# Unique edge types	4	4	4
# Graph nodes	2,551,821	1,981,292	570,529
# Graph edges	7,071,122	5,448,259	1,622,863
# Labels	184,265	90,321	93,944

Table 1: A summary of our dataset.

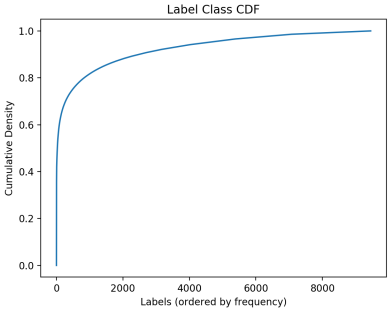

Figure 4: CDF of label frequencies in the dataset.

large label set, we throw out all labels that are not in the top 20 most common labels, leaving us with roughly 50% of the total labels in the training dataset. The CDF of label frequencies can be seen in Figure 4, showing the long tail of infrequent labels. To give a better idea of the type labels that we predict, the top 5 most common types in the training set are: `string`, `number`, `boolean`, `any[]`, and `string[]`.

Train-Test split. We made sure to do our train/test split on the dataset not based on file graphs, but on entire source projects, since we are primarily interested in how this approach performs on code that it has never seen before. Files within the same project do tend to have similar, even identical code snippets, so we wanted to rule out any potential false positives from memorizing such sub-graphs. This is, however, a conservative model of a real world system: a type predictor built into an IDE would have access to other files within the same project that the user has written, and would be able to base predictions on that. Because of this, we have a separate set of weights for computing the initial node and edge embeddings, so that a model can refine its training on graphs specific to one type of project, while keeping the same node/edge embeddings and predictors, essentially performing a version of the standard NLP-style transfer learning, but for program graphs.

3.2 Graph construction.

Because "graph neural nets" are such a general framework, the inductive bias for our solution comes primarily from how the input graph is constructed. Specifically, we must decide on the nodes and edges that comprise our generated graph. We choose to include the full set of nodes in each program's AST as the nodes in the input graph, using the node's syntactic token type as its embedding. This allows us to include all potentially relevant structure: we have not only the source level tokens in the source file, but also the abstract structures that those tokens form.

Graph edges are significantly more complicated. We want to take advantage of all of the important structure that we know about in the program, while also trying to avoid creating as many useless edges as possible, both to help with computation and to induce a stronger prior over the solutions we think might be valid. Specifically, we take advantage of our knowledge of the important local and nonlocal interactions between various parts of programs, similar to how convolutional networks assume some strong relationship between local pixels [7]. There are three specific priors we incorporate into the edges in our generated graph:

Nodes near a variable in the AST reflect something about that variable's type.

```
let x = y * 2;
```

A human trying to infer the type of the variable `x` in the above code may decide that since the result of multiplying something by a number is almost always a number, `x` is probably a number. This corresponds exactly to two types of edges that we include in our generated graph, an `AstChild` edge from parents to their child nodes in the AST, and an `AstParent` edge from children to their parent in the AST.

Nodes near a variable in the file reflect something about that variable's type.

```
let x = 1;
let y = x;
...
let x = "foo";
```

It is clear that both ordering and locality matter in determining the type of `y`: in the AST, the two statements assigning to `x` are equidistant and directionally indistinguishable, but the actual *token stream* of the file reflects that the first assignment to `x` precedes the assignment to `y`, meaning that it is more likely to affect the type of `y`. It is also closer, meaning the two statements are even more likely to be related. Because of this, we include two more types of edges in the induced graph, a `TokenNext` edge from a token to its neighbor in the source file, and a `TokenPrev` edge which is the reverse.

Nodes that use or define the same variable give information about each other.

```
let x = 1;
...
let y = a + (b + (... + x));
```

Although the two statements may be arbitrarily distant in both the AST and the source file, they are clearly related,

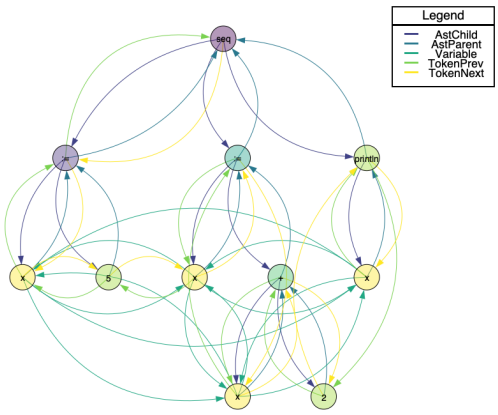

Figure 5: Generated graph for a simple example

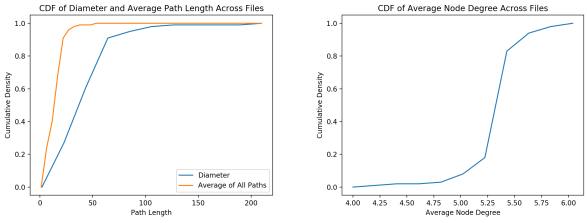

Figure 6: A summary of key dataset statistics, showing the CDFs of the average path length, the diameter of the graphs, and the average node degrees of the graphs in the dataset.

in that `x`'s type influences `y`'s type. Because of this, we include one final type of edge, a `Variable` edge, between any usages of the same variable.

Example. With all of the above nodes and edges included, we generate something like the graph shown in Figure 5 for the following code:

```
let x := 5;
x = x + 2;
console.log(x);
```

Some high level statistics about the graphs we generate (which we will refer to later to diagnose behavior of our GNN) can be seen in Figure 6.

3.3 Embeddings.

In the program graph, a node's initial embedding is a one-hot vector encoding its syntactic AST type (e.g. PLUSOPERATOR, ARRAYACCESS). Similarly, an edge's embedding is a one-hot vector encoding its edge type. We keep the top 60 node types, and consolidate the rest into an UNK token, to avoid sparsity on the bottom end of the node type distribution (the distribution of AST node types in our dataset can be seen in Figure 7; the top 60 node types correspond to 95% of all nodes). We do not do the same with edge types, since by construction all edge types are densely populated.

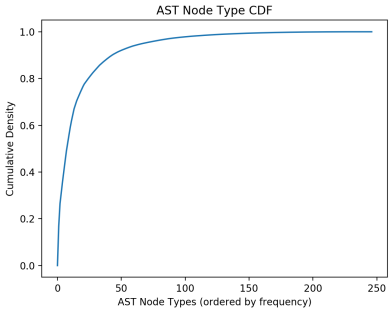

Figure 7: CDF of AST type frequencies in the dataset.

3.4 Architecture.

The graph architecture we explore is fairly simple, with minimal observed effects from architecture tuning. The core of the architecture is the graph neural net, described in Section 3.2. To use this, we perform some minor pre- and post-processing on the data: we have a one-hidden-unit fully connected layer to transform the one-hot encodings into the initial node and edge embeddings, and after running the graph, we similarly have a one-hidden-unit fully connected layer to transform the final latent state into the predictions. These two fully connected layers are the parts held constant for the transfer learning approach mentioned above, in Section 3.1.

3.5 Implementation.

We implemented our type prediction algorithm using using DeepMind's Graph Nets framework [8], along with a significant amount of custom TensorFlow to collect metrics and perform experiments. Our implementation, along with minor extensions to the Graph Nets framework, can be found online[1].

4 Experiments

We investigate the following questions:

- **Q1.** How well does encoding our inductive bias as a GNN perform in tasks which have a graph-like structure?
- **Q2.** How does the nature of structural information passed along the edges of a GNN affect its performance?
- **Q3.** How can the number of message passing iterations be learned from data, rather than tuned manually?

Experiment 1: Representation using GNN. We investigate whether the inductive bias captured in the structure of graph neural networks helps in prediction tasks on graph structured data. To determine whether our GNN architecture models this prediction task better than alternative methods, we set up a number of baselines to compare against. We investigated the following models:

- **Our model: GNN.** We model the type inference problem using GNNs. Each node represents a node in the program's AST. We infer the types on the nodes associated

[1] https://github.com/shashank-srikant/6.867_term_project/blob/master/src/graph_neural_net/nn.py

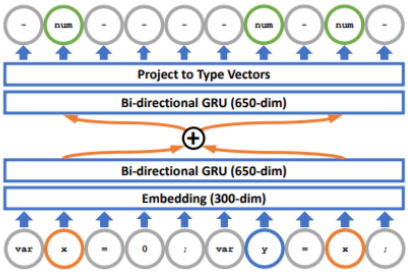

Figure 8: Aggressive baseline: A bi-directional RNN designed by Allamanis *et al.* [2]. It models type inference as a sequence prediction task in NLP.

with variables. This is the model we compare all baselines against.

- **Baseline 1: Naïve.** We design a naïve baseline to compare performance. In this model, we perform a majority vote on the label distribution in our train set, using the most frequently occurring label as the type predicted for any unseen variable (respectively, the top k most frequently occurring labels for the top k predictions). This provides a maximum likelihood estimate.
- **Baseline 2: Intermediate.** Increasing the complexity of our baseline models, we evaluate a naïve approximation of a graph net. Here, we construct a logistic regression model over a feature vector encoding information about the other variables used in any given variable's definition. Specifically, for each definition of a variable `let v = expr`, we concatenate the following feature vectors:
  - Consider each variable x used in `expr`. Define the 'path' from v to x to be the string of AST node types encountered in a walk along the AST from v to x. We construct a feature vector with the sum of the one-hot vectors keyed on 'path' for each variable x used in `expr`.
  - A multi-hot vector representing each unique AST node type appearing in `expr`.

  We enumerated 6183 unique paths and 89 unique AST node types in the training set. This resulted in a $524507 \times (6183 + 89)$ data matrix, which was used to learn a regularized Logistic Regression model. This baseline captures a *bigram variable dependence model*, where only the dependency effect of other variables used in a given variable's definition are considered, without the full dependency information flow provided by GNNs.
- **Baseline 3: Aggressive.** We use the bi-directional neural network architecture designed by Allamanis *et al.* [2]. Their architecture does not utilize any of the graphical properties of a program, and instead relies on signals obtained by just token information. Their approach is typical of sequence prediction tasks in NLP. Figure 8 describes their architecture.

Performance metric. For each of these models, we compute two measures of accuracies: P@1, P@5. A P@k accuracy corresponds to percentage of the model's top k predictions containing the true label of a given item. This is a standard accuracy measure used in classification tasks.

Experiment 2: Edge ablation. To determine whether our graph structure was chosen correctly, we also performed a set of ablation studies. Specifically, we ran experiments using the optimally chosen hyperparameters where we entirely deleted each class of edges (AST edges, TOKEN edges, and VARIABLE edges) from the graph, and compared performance to the original.

Experiment 3: Number of iterations. A major shortcoming in the graph net framework is the issue of determining how many iterations of message passing to run. In theory, because of the universal approximation properties of neural nets, the optimal number of iterations is any number larger than the diameter of the graph (length of the longest path in the minimum spanning tree), so that each node can make a fully informed decision knowing the entire graph topology. However, this is not realistic: as shown in Figure 6, the maximum diameter of any of the graphs is above 200, meaning that we would have to run the graph neural net for an intractable number of iterations to be able to get this idealized result. Instead, we approximate the result by running fewer iterations, theoretically losing out on certain classes of decisions, although in doing so further enforcing our inductive bias by effectively ruling out all distant communication between nodes. It is not immediately clear how we should choose the NITER hyperparameter though: since graph nets give no formal guarantees that the results monotonically improve with the number of iterations, we have to select the number of iterations through some hyperparameter search. We ran two experiments to try to select the ideal settings:

- ***Bayesian Optimization* based approach.** We were inspired by data-driven approaches to hyperparameter search problems, for cases where running experiments can be prohibitively expensive. In particular we were inspired by [15], which suggests using a Bayesian Optimization based approach with an underlying Gaussian Process to pick hyperparameters, using the expected improvement metric (expectation of the decrease in the desired metric over the current best known hyperparameter) to explore the hyperparameter space.
- ***Iteration Ensemble* approach.** Our second attempt at a hyperparameter search followed a more directly optimization based approach. Rather than trying to pick one specific ideal number of iterations, we ran an ensemble over several choices of iteration counts, using a learned weighting to linearly combine all of their predictions in the last step to produce the final result. Importantly, we ran this iteration ensemble on *one single graph net*; that is, we combined each of the intermediate predictions of running a single graph neural net. We considered this for two reasons: primarily, it was more computationally efficient (running large ensembles of graph nets was not possible on the machines we had access to), but it also enforced one of the inductive biases we hoped to see in a solution: intermediate steps of the graph net message passing algorithm should roughly correspond to finer and

finer approximations of the posterior distribution, meaning that all intermediate steps should be valid (if unrefined) predictions. We also hoped to see situations where weights on the upper or lower end of the ensemble would drop away, signifying that we should shift (increase or decrease) the space of NITER that we are searching over.

5 Results and Discussion

Experiment 1. The results of this can be seen in Table 2. The best results we achieved, using the graph neural net with NITER $= 2$ and including all AST, VARIABLE, and TOKEN edges had P@1 and P@5 accuracies of 66% and 96% respectively.

Regarding baselines, we see that the *intermediate* baseline performs as expected. It predicts *number* and *string* types well, and fails at predicting any other type accurately. Analyzing the model's selected features reveals node types like $+$ are highly weighted, which correspond to operations specific to *number*s and *string*s. We leave a detailed analysis of these learned features to future work.

The *aggressive* bi-directional RNN baseline, which was the best model trained by Allamanis *et al.* [2], performs in the ballpark reported in their work (slightly worse than [2], where the variance is explained by the smaller size of our dataset). It has a P@1 accuracy of 50.6%, 16% lower than the GNN's performance.

These results confirm that the graph neural net framework, along with the assumptions we made about the graph structure, work well to model the problem: our solution roughly halves the error rate from random guessing, and significantly outperforms all other baselines.

GNN model diagnosis. We also include a sampling of the false positive and negative rates on the test set in Table 4. It shows only two significant outliers, the number of false predictions on the `string` type and the number of missed predictions of the `number` type. Given that some of the most common operators on numbers and strings are syntactically identical in TypeScript ($+$ adds numbers and concatenates strings), it is unsurprising that numbers would get interpreted as strings, especially considering that the MLE guess when the two are syntactically indistinguishable would be `string`; this intuition has been confirmed by checking that the majority of missed `number` predictions are predicted as `string`s.

Experiment 2. The results of the ablation studies can be seen in Table 3. These experiments confirm our assumptions, that each of the types of edges we selected help in solving the type inference problem. We can additionally see the relative benefit of each of the edge types: the syntactic locality induced by the TOKEN edges seems to matter more than the AST edges, which in turn matter more than the VARIABLE edges. While it is somewhat surprising that the VARIABLE edges are the least important, since type inference is based in part on variable usage, this can be explained in part by the relatively small number of message passing iterations: there is no immediate information gained from a linked variable, only from that linked variable's neighbors, so more iterations would likely be needed to see as much benefit from the VARIABLE edges.

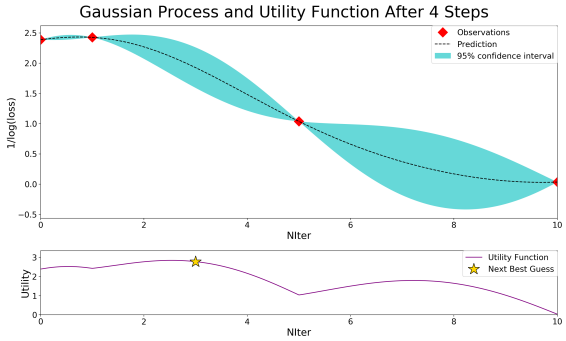

Figure 9: Partial result of Bayesian Optimization on NITER (after 4 samples)

Experiment 3. Bayesian Optimization. The metric we chose to optimize over was the train loss after 3 epochs. Selecting the best performing number of iterations with this particular metric gave us confidence that the network had actually made progress in learning significant features of the dataset, while also being sure not to overfit to our validation set. Partial results of this experiment can be seen in Figure 9, which shows the confidence bounds on the number of iterations to use in the range $[0, 10]$. These bounds gave us a satisfactory coverage of enough message passing to propagate through the average path in the graph (c.f. Figure 6), while still being computationally tractable to train. After 5 interactions with the Gaussian process, we were confident that $n = 1$ iteration gave the best training loss after 3 epochs, although we used $n = 2$ in our experiments, since it seemed to do marginally better on the validation set.

We were slightly surprised by the relatively low number of iterations found as ideal by the Bayesian Optimization. Ultimately, we believe that this is probably an issue with the amount of time that we trained for: since a message passing graph net unfurls to look similar to a RNN, it encounters the same vanishing gradient problem, magnified by the large fanout of each node (Figure 6 shows an average node degree above 5). Were we to train for significantly longer, we may be able to deal with the smaller gradients and the more complex loss landscape of a larger network, but unfortunately training the larger NITER networks for longer on our machines was computationally infeasible – we leave this deeper parameter exploration to future work.

Iteration Ensemble. Ultimately, this experiment did not work as well as we hoped, and our best results were achieved by just using a single number of iterations learned through the Bayesian Optimization approach. The linear combination weight vector changed only marginally from its initial value, and the prediction quality was no better than that of using a fixed number of iterations. We believe this happened for one primary reason: one or two iterations

	Model	P@1	P@5
Our model	**GNN**	0.66	0.96
Baselines	**Naïve** (Majority vote)	0.31	0.87
	Intermediate (Dependency counts)	0.44	0.93
	Aggressive (Bi-directional RNN)	0.50	*

Table 2: Results of model comparison against various baselines, ordered by their model complexity. These are results on the test-set, on the top 20 most frequent labels in the train-set. P@1, P@5 correspond to the accuracy of top-1 and top-5 predictions matching the exact label.

Ablated Model	P@1	P@5
Full GNN model	0.66	0.96
AST edges removed	0.55	0.93
Token edges removed	0.43	0.94
Use-Define edges removed	0.61	0.92

Table 3: Results of our ablation study.

Label	#Correct	#Missed	#FP
`string`	6374	1021	4216
`number`	5501	4006	881
`boolean`	1610	850	1089
`any[]`	367	240	226
`string[]`	130	321	141
`() => void`	606	279	190
`A`	0	0	301
`undefined`	37	58	179
`() => string`	91	112	111

Table 4: Top 10 test labels (23551 total predictions)

were good enough, as shown by the Bayesian Optimization approach. Any iterations beyond that did not refine the search, and since we did not penalize their existence through any regularization (only the wrongness of their predictions), they stayed around as unnecessary artifacts that did not get pruned.

6 Conclusion

Our results demonstrate that for learning tasks on graph-based structures, the inductive bias introduced by GNNs offers a significant benefit over other modeling choices. For the specific task of type prediction on our dataset, we show that GNNs perform better than state of the art modeling choices like bi-directional RNNs. We also confirm through experiments that the right choice of edges in the graph has a significant effect on the performance of the learning task. Despite these results, we have not yet validated one of our initial assumptions, that a graph neural net is actually approximating some underlying probabilistic graphical model. We believe there is an important larger research question about how to formalize GNNs through the lens of probabilistic graphical models. Theoretically and experimentally proving (or disproving) their correspondence is an open question, and would provide much deeper insight into these results.

7 Task split-up

- **Alex.** Graph neural network implementation, analysis, report writing.
- **Katie.** Graph neural network implementation, analysis, report writing.
- **Shashank.** Typescript parsing and graph generation, analysis, report writing.

References

[1] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[2] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 152–162, New York, NY, USA, 2018. ACM.

[3] Pavol Bielik, Veselin Raychev, and Martin Vechev. Programming with" big code": Lessons, techniques and applications. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[4] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[5] Shashank Srikant and Varun Aggarwal. A system to grade computer programming skills using machine learning. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1887–1896. ACM, 2014.

[6] Qi Liu, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L Gaunt. Constrained graph variational autoencoders for molecule design. *arXiv preprint arXiv:1805.09076*, 2018.

[7] Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. *CoRR*, abs/1506.05163, 2015.

[8] DeepMind. Graph nets. `https://github.com/deepmind/graph_nets`, 2018.

[9] Jure Lescovec, William L. Hamilton, Rex Ying, and Rok Sosic. Representation learning on networks. Tutorial at WWW, 2018.

[10] R. Kinderman and S.L. Snell. *Markov random fields and their applications*. American mathematical society, 1980.

[11] Judea Pearl. *Causality: Models, Reasoning and Inference*. Cambridge University Press, New York, NY, USA, 2nd edition, 2009.

[12] Yair Weiss. Correctness of local probability propagation in graphical models with loops. *Neural Comput.*, 12(1):1–41, January 2000.

[13] Anand Rangarajan, Rama Chellappa, and Anand Rangarajan. Markov random field models in image processing, 1995.

[14] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, July 1989.

[15] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, pages 2951–2959, USA, 2012. Curran Associates Inc.